Vol.2017-SE-195 No.27
2017/3/13

テストを用いた形式仕様と実装の不整合検出

山本　大祐[1,a)]　大森　洋一[2]　荒木　啓二郎[2]

概要：形式手法を用いた開発では形式仕様記述言語を用いて記述した仕様を元に実装を行なっていくが，様々な不具合が生じることがある．この不具合の原因として仕様と実装の間で整合性が満たされていない可能性がある．これは仕様と実装の間で扱うことのできるデータ構造などの違いから生まれるものである．本研究の目的は，形式仕様記述を用いたソフトウェア開発において，開発の効率化のために仕様と実装の間での整合性を乱す具体的な原因をテストを用いて明らかにすることである．本研究では，既存の形式仕様を用いて，ソフトウェアを実装し，その仕様に対してテストケースの生成を行うとともに，仕様のテストと意味的に等価なテストを実装に対しても適用する．そのテスト結果を元に仕様と実装の不整合について修正方法を考察する．

キーワード：証明, 検証付きプログラミング, 形式手法の適用, テスト技法, 保守技術,VDM,Java

Identification of Software Inconsistencies between Specification and Implementation using Semantically Shared Tests

Abstract: Software development using formal methods is driven by a mathematical model described in formal modeling language. It leads from the requirements through the implementation, but some inconsistencies might be found during the procedure. The inconsistency comes from the difference of execution models and limitation of resources. We propose a method to identify some types of inconsistencies between the specification and implementation using semantically shared tests. A case study about an existing formal specification in VDM and new implementation in Java depicts the testing sequence and the update alternatives in maintenance.

Keywords: Programs with proofs and certifications,Applitcation of formal methods,Testing and Haimtenance,VDM

1. はじめに

1.1 研究の背景

ソフトウェア開発プロセスは仕様記述の前後で開発対象の要求分析やそれに対する仕様策定などの上流工程と，上流工程の結果を受けて，開発対象を実装していく下流工程に分けることができる．上流工程において自然言語を用いて仕様を記述すると，自然言語は曖昧な表現を含むため，欠陥を埋め込みやすい．この曖昧な仕様をもとに開発を進めると，下流工程で欠陥を発見することになり，欠陥を取り除くための手戻りが発生することになる．手戻りには作業及び時間的コストがかかるため，手戻りが起こることは望ましくない．手戻りを防ぎ効率よく開発を行うために，システムの要求を正しく理解すること，また記述された仕様が十分かどうか検証することが求められている．その方法の一つとして形式仕様記述がある．形式仕様記述とは形式仕様記述言語と呼ばれる数学的に厳密な意味論を持つ言語を使用してシステムを記述することにより，仕様に内在する自然言語による表現の曖昧さを排除し，要求の正しい理解を可能にするとともに，システムをモデル化することにより，期待される性質を満たしているかどうかを数理的な検証を可能とする技術である．

しかし，この形式手法を用いても完全に手戻りを防ぐことができるわけではない．仕様と実装の間では様々な要因から，不整合が生じる．不整合を防ぐために，妥当性の検証方法として証明とテストの２つがある．これまでの形式手法適用では数学的証明や段階的詳細化による方法が用いられてきた．しかし，そうした証明による方法は数学的な

[1] 九州大学工学部電気情報工学科,
Motooka744,Nishi,Fukuokashi 819-0375,Japan
[2] 九州大学システム情報科学研究院,
Motooka744,Nishi,Fukuokashi 819-0375,Japan
[a)] yamamoto.daisuke@nanotsu.ait.kyushu-u.ac.jp

知識を必要とする．それに対してテストを用いると，

- 専門的な知識が必要なく，誰にでも結果がわかる．
- ユニットテストによる繰り返し実行を自動化できる．

といった利点がある．ただし，テストを用いた妥当性の検証では，妥当性の検証はできても不整合の原因まではわからない．この原因を事前に把握し実装の段階で注意することで，手戻りを減らし形式手法を用いた開発をより効率的に行うことができる．

1.2 研究の目的

実行可能な形式仕様と実装で同じ意味を持つテストを実行することにより，以下の点を明らかにするソフトウェア開発手法を確立し，事例により検証する．

(1) 実装が仕様を満たしていることをテストにより示す．
(2) テスト結果の違いから，仕様と実装の不整合を見つける．

1.3 研究の方針

本研究の方針としては，VDM 開発支援ツール Overture-tool の Web サイト [3] で提供されている銀行 ATM を題材とした形式仕様記述言語 VDM-SL でモデル化された仕様を元にユニットテストのテストケースを C1 カバレッジを用いて生成する．その後，Java 言語で書かれたプログラムを実装する．その過程で，仕様に対するテストケースと等価なテストを実装に対しても適用する．仕様と実装でのテスト結果を元に不整合についての考察を行う．さらに，不整合から生まれる問題点を仕様へとフィードバックする．

2. 形式手法 VDM

形式手法とは，形式仕様記述言語を用いることで仕様を厳密に記述する方法である．日本語は同じ単語であっても，文脈や解釈が違うことがあり，日本語で仕様を記述した場合，その仕様には曖昧さが残る．形式手法を用いることで，日本語の曖昧な表現を排除し解釈を統一することができる．形式仕様記述言語は数学的に意味付けられた言語であり，検証したい性質によって表現可能な数理体系は様々であり，それぞれに応じた形式手法がある．形式仕様記述言語で書かれた仕様はソフトウェア・システムの数理的モデルであり，そのモデルを検証することによって，実際のプログラムを作成する前に，仕様に内在する欠陥を見つけ出すことができる．これによりソフトウェアの品質を高めることができる．

2.1 VDM

VDM(Vienna Development Method) は 1960 年代に IBM のウィーン研究所において開発された形式手法の 1 つである．VDM の意味論は一階述語論理や集合論が基になっている．形式仕様記述 VDM には「VDM-SL」，「VDM++」，「VDM-RT」という形式仕様記述言語がある．VDM の意味論において実行可能な VDM 仕様は定義された文法を用いて，対象となるシステムのモデル化及び記述を行い，記述した仕様にテストケースを与えて動作させることでテストができる．このテストを用いることで，仕様の妥当性を検証することができる．本研究では VDM-SL を使用する．VDM-SL は VDM の形式仕様記述言語の 1 つである．VDM-SL は，1996 年に [ISO/IEC13817-1](International Organization for Standardization) にて標準化されている．

2.2 VDM 開発支援ツール Overture

Overture tool は Aarhus 大学の Peter Gorm Larsen 教授を中心とするコミュニティによって開発されている VDM の記述，検証，実行を行うオープンソースの統合開発環境であり，最新の VDM 規格をサポートする．本研究では Overture tool の以下の機能を使用した．[3]

- 仕様の構文・型チェック機能
- 実行可能仕様のインタプリタとデバッグ機能
- 実行可能仕様のコードカバレッジ計測機能
- VDM-SL の清書機能

3. 実験の手順

3.1 ATM モデル概要

本研究の題材である銀行 ATM(Automatic Teller Machine) モデルは実際の ATM システムのソフトウェア開発に用いられた仕様を教材として公開できるように抽象化したものである．ATM モデルの機能に関して管理者，ユーザーに分けて使用可能な機能を記述する．

管理者側の機能

- 新規口座の開設
- カードの使用停止

ユーザー側の機能

- 引き出し
- 残高照会

ATM モデルのクラス図を**図 1** に示す．

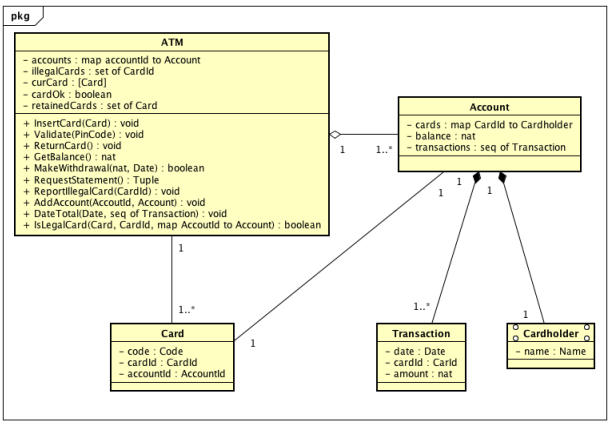

図 1 ATM モデルクラス図

ATM モデルのユースケース図を**表 2** に示す.

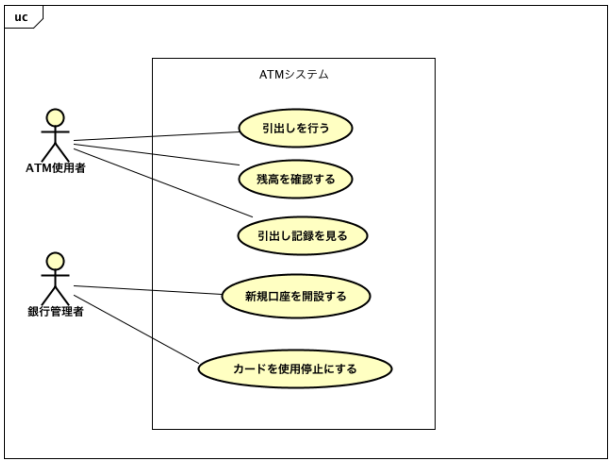

図 2 ATM モデルユースケース図

3.2 VDM-SL で記述された ATM モデル

前節で説明した ATM モデルは形式仕様記述言語 VDM-SL で記述されている.

3.2.1 型定義

型として定義されているものの中で合成型の構成に用いられる ATM モデルにおける基本型の説明を**表 1** に示す.

表 1 ATM モデルの基本型定義

名前	型またはシグネチャ	説明
AccountId	nat	アカウントの ID
Name	seq of char	アカウントの所有者名
CardId	nat	カード番号
Code	nat	設定した暗証番号
PinCode	nat	入力した暗証番号
Date	seq of char	日付

次に ATM モデルにおける合成型の説明を以下に示す.

```
types
  --アカウント情報
  Account :: cards       : map CardId to Cardholder
                         --カードIDから所有者への写像
             balance     : nat
                         --残高
             transactions    : seq of Transaction
                             --アカウントの引出し記録
    --不変条件
  inv
  account==TransactionsInvariant(account.transactions);
  --引出し情報
  Transaction :: date : Date   --引出し日付
                 cardId : CardId--引出したカード番号
                 amount : nat;   --引出額
  --カード情報
  Card :: code     : Code     --暗証番号
          cardId   : CardId   --カードID
          accountId : AccountId;--アカウントID
  --カード所有者名
  Cardholder :: name : Name;
```

アカウント情報の不変条件について説明する.

- 引出し記録に含まれる全ての日付における引出し合計が dailylimit=2000 を超えない.

3.2.2 状態定義

ATM モデルの状態変数について表**??**に示す.

```
state System of
  accounts : map AccountId to Account
           --AccountIdからAccountへの写像
  illegalCards : set of CardId
            --使用不可なCardIdの集合
  curCard : [Card]
            --現在のCard情報
  cardOk : bool
            --引出し可能かどうかを示す
  retainedCards : set of Card
            --ATMに格納されたCardの集合
--不変条件
 inv mk_System(accs,-,curCard,cardOk,-) ==
   (curCard = nil => not cardOk) and
   (forall id1, id2 in set dom accs &
   id1 <> id2 =>
   dom accs(id1).cards inter dom accs(id2).cards = {})
--初期状態
 init s == s = mk_System({|->},{},nil,false,{})
end
```

状態変数の不変条件について説明する.

- curCard が空ならば cardOk=false である.
- accounts の定義域の全てのキーにおいて対応する Ac-

count の cards の定義域に同じものが存在しない
状態変数の初期状態について図 3 に示す．ここでの初期状態は ATM の最初の起動時の状態である．

```
accounts    :{|->}  利用者が登録されていない
illegalcards :{}    使用不可なカードがない
curCard     :nil    カードがATMに挿入されていない
cardOk      :false  引き出しは不可能である
retainedCards  :{}    格納されたカードはない
```

図 3　ATM モデル初期状態

3.2.3　定数定義

```
values
 dailyLimit : nat = 2000;
```

定数として定義されているのは dailyLimit である．これは 1 日の引き出し金額の限度を設定しており，2000 と定義する．

3.2.4　操作・関数定義

ここでは各操作・関数のユースケースとの対応を述べる．まず，節 3.1 で示した．ユースケース図を詳細化したものを図 **4** に示す．

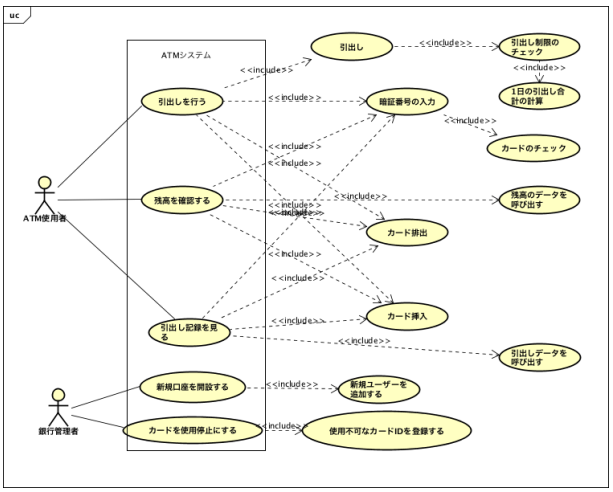

図 4　ATM モデルユースケース図 2

各操作・関数とユースケースの対応を**表 2** に示す．

表 2　関数・操作のユースケースとの対応

ユースケース	対応操作・関数名
カードの挿入	InsertCard
カードの排出	ReturnCard
暗証番号のチェック	Validate
カードのチェック	IsLegalCard
残高のデータを呼び出す	GetBalance
引出し	MakeWithdrawal
引出し制限のチェック	TransactionInvariant
1 日の引出し合計の計算	DateTotal
引出しデータを呼び出す	RequestStatement
新規ユーザーを追加する	AddAccount
使用不可なカードを登録する	ReportIllegalCard

3.3　テスト作成手順

本研究では VDM-SL の統合開発環境である Overturetool の web サイトで提供されている cashdispencer.vdmsl を仕様として，仕様に対するテストケースの生成を行なった．この際のテストは作成基準として C1 カバレッジを用いたユニットテストである．その後，Java を用いて実装を行う過程で，仕様に対して行なったテストケースと等価なテストを Java に対しても適用し，結果の比較・考察を行なった．

3.3.1　C1 カバレッジ

ソフトウェアに対してテストを行うにあたり，考えられうる全ての入力をテスト (全数テスト) できれば，ソフトウェアの品質が保証できることになる．しかしそのテストケースは膨大で現実的ではなく，全数テストは不可能である．そこで全数テストと同様の網羅性を維持しつつ，テストケースの数を削減しようというテスト技法がある．網羅性を基準としたテスト基準の 1 つが C1 カバレッジである．C1 カバレッジは分岐条件網羅と呼ばれ，関数内での判定条件がどれだけ行われたかを用いて評価する．今回は事前条件や if 文による分岐を判定条件として，テストケースを作成した．

3.4　ATM モデルに対して生成したテストケース

ここでは前述した ATM モデルに対して適用したテストケースについて述べる．以下に各操作・関数に対するテストケースの概要を述べる．ただし，仕様には記述されているものの実装上不要と判断した Encode，Sum,Len に対するテストケースは作成していない．テストケース作成において，各操作・関数の事前条件と if 文の分岐判定条件を C1 カバレッジで分岐網羅するようにテストを作成した．

3.4.1　InsertCard

表 3　InsertCard に対するテスト

テスト名	内容	期待結果
Insert_test0	カードを 1 枚挿入	正常終了
Insert_test1	カードを 2 枚連続で挿入	事前条件違反

3.4.2 Validate

表 4 Validate に対するテスト

テスト名	内容	期待結果
Vali_test0	カード未挿入で Validate	事前条件違反
Vali_test1	使用不可なカードで Validate	<Retained>
Vali_test2	間違った暗証番号の入力	<PinNotOk>
Vali_test3	正しい暗証番号の入力	<PinOk>
Vali_test4	cardOk が true での Validate	事前条件違反

3.4.3 ReturnCard

表 5 ReturnCard に対するテスト

テスト名	内容	期待結果
Return_test0	カードの排出	正常終了
Return_test1	カード未挿入でカード排出	事前条件違反

3.4.4 GetBalance

表 6 GetBalance に対するテスト

テスト名	内容	期待結果
Get_test0	カード未挿入	事前条件違反
Get_test1	暗証番号未認証	事前条件違反
Get_test2	使用不可なカード	事前条件違反
Get_test3	正常動作	正常終了

3.4.5 MakeWithdrawal

表 7 MakeWithDrawal に対するテスト

テスト名	内容	期待結果
Make_test0	カード未挿入で引き出し	事前条件違反
Make_test1	暗証番号未認証での引き出し	事前条件違反
Make_test2	使用不可なカードで引き出し	事前条件違反
Make_test3	引出額が負の引き出し	事前条件違反
Make_test4	正常動作	true
Make_test5	1 日の引き出し条件を超える	false
Make_test6	残高以上の引き出し	false
Make_test7	合計が int 型の対応値を超える	false
Make_test8	引出し記録数が 100 を超える	true

※ Make_test8 は実装に対してのみ追加で作成し適用した.

3.4.6 RequestStatement

表 8 RequestStatement に対するテスト

テスト名	内容	期待結果
Req_test0	カード未挿入	事前条件違反
Req_test1	暗証番号未入力	事前条件違反
Req_test2	使用不可なカード	事前条件違反
Req_test3	正常動作	正常終了

3.4.7 IsLegalCard

表 9 IsLegalCard に対するテスト

テスト名	内容	期待結果
Islegal_test0	使用可能カードの判定	true
Islegal_test1	使用不可なカードの判定	false
Islegal_test2	登録されていないカードの判定	false
Islegal_test3	アカウントに存在しないカードの判定	false

3.5 ReportIllegalCard

表 10 ReportIllegalCard に対するテスト

テスト名	内容	期待結果
Repo_test0	カード ID1000 を使用不可に	正常終了

3.5.1 AddAccount

表 11 AddAccount に対するテスト

テスト名	内容	期待結果
Add_test0	新規口座の開設	正常終了
Add_test1	既存のアカウント ID で新規開設	事前条件違反

3.6 Java での実装

ここでは Java での実装について記述する.

3.6.1 クラス

Java でのクラス図を図 5 に示す.

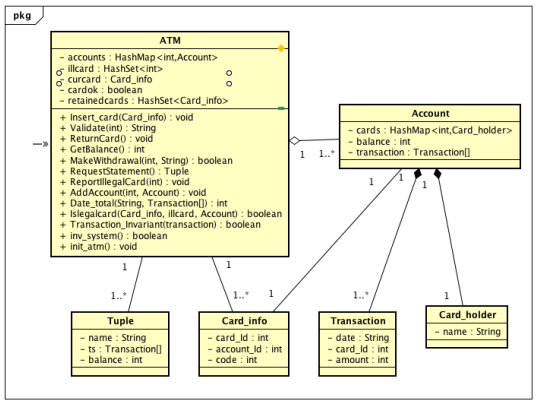

図 5 ATM モデルクラス図

VDM-SL と Java でのクラスと対応を表 12 に示す.

表 12 仕様と Java のクラス対応

仕様でのクラス名	Java でのクラス名
ATM	ATM
Card	Card_info
Account	Account
Transaction	Transaction
Cardholder	Card_holder
未定義	Tuple

RequestStatement の戻り値として複数の値を返すために, 新たなクラス Tuple を定義した.

3.6.2 型定義

VDM-SL と Java どちらにも定義されている型は同じ型を用いて定義した．Java で定義されていない型に関して異なる型を用いて実装を行なったものを**表 13** に示す．

表 13 仕様と Java の異なる型対応

仕様での型	Java での型
nat	int
seq of char	String
seq of Transaction	Transaction[]

3.6.3 操作・関数

仕様と Java での操作・関数の対応を**表 14** に示す．

表 14 操作・関数対応

仕様での操作名	Java でのメソッド名
InsertCard	Insert_card
Validate	Validate
ReturnCard	ReturnCard
GetBalance	GetBalance
MakeWithdrawal	MakeWithdrawal
RequestStatement	RequestStatement
ReportIllegalCard	ReportIllegalCard
AddAccount	AddAccount
DateTotal	Date_total
IsLegalCard	Islegalcard
初期化	init_atm
システムの不変条件	inv_system
Transaction の不変条件	Transaction_Invariant

Java での実装で関数を 3 つ追加した．

- init_atm
- inv_system
- Transaction_Invariant

である．init_atm はシステムの状態を初期化する関数である．inv_system はシステムの不変条件が満たされているかどうかを判定し，bool 値を返す関数である．Transaction_Invariant は Transaction に対する不変条件を満たされているかどうかを判定し，bool 値を返す関数である．不変条件は各操作・関数内の事前条件・事後条件として記述することで実装を行なった．事前条件・事後条件は各関数に assert 文として記述した．事前条件・事後条件の対応を**図 6** 示す．

```
public int sum(int[]){
 assert 事前条件;
 処理
 assert 事後条件;
 return 結果;
}
```

図 6 事前条件・事後条件の実装例

Java の文法仕様の制約により事後条件は return 文の直前に記述している．

4. 考察

4.1 実行結果

テスト結果は実行可能な VDM-SL と Java による実装を対比させて示す．各操作・関数に対するテストの実行結果を操作・関数ごとに表 15〜表 23 に示す．

4.1.1 InsertCard

表 15 InsertCard に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Insert_test0	正常終了	正常終了	正常終了
Insert_test1	事前条件違反	事前条件違反	事前条件違反

4.1.2 Validate

表 16 Validate に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Vali_test0	事前条件違反	事前条件違反	事前条件違反
Vali_test1	事前条件違反	事前条件違反	事前条件違反
Vali_test2	<Retained>	<Retained>	<Retained>
Vali_test3	<PinNotOk>	<PinNotOk>	<PinNotOk>
Vali_test4	<PinOk>	<PinOk>	<PinOk>

4.1.3 ReturnCard

表 17 ReturnCard に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Return_test0	正常終了	正常終了	正常終了
Return_test1	事前条件違反	事前条件違反	事前条件違反

4.1.4 GetBalance

表 18 GetBalance に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Get_test0	事前条件違反	事前条件違反	事前条件違反
Get_test1	事前条件違反	事前条件違反	事前条件違反
Get_test2	事前条件違反	事前条件違反	事前条件違反
Get_test3	正常終了	正常終了	正常終了

4.1.5 MakeWithdrawal

表 19 MakeWitthdrawal に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Make_test0	事前条件違反	事前条件違反	事前条件違反
Make_test1	事前条件違反	事前条件違反	事前条件違反
Make_test2	事前条件違反	事前条件違反	事前条件違反
Make_test3	事前条件違反	事前条件違反	事前条件違反
Make_test4	true	true	true
Make_test5	false	false	false
Make_test6	false	false	false
Make_test7	false	false	true
Make_test8	true	未実施	false

4.1.6 RequestStatement

表 20 RequestStatement に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Req_test0	事前条件違反	事前条件違反	事前条件違反
Req_test1	事前条件違反	事前条件違反	事前条件違反
Req_test2	事前条件違反	事前条件違反	事前条件違反
Req_test3	正常終了	正常終了	正常終了

4.1.7 IslegalCard

表 21 IslegalCard に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Islegal_test0	true	true	true
Islegal_test1	false	false	false
Islegal_test2	false	false	false
Islegal_test3	false	false	false

4.1.8 ReportIllegalCard

表 22 ReportIllegalCard に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Repo_test0	正常終了	事前条件違反	事前条件違反

4.1.9 AddAccount

表 23 AddAccount に対するテスト実行結果

		実行結果	
テスト名	期待結果	VDM-SL	Java
Add_test0	正常終了	正常終了	正常終了
Add_test1	事前条件違反	事前条件違反	事前条件違反

4.2 考察

4.2.1 テストから検出された不整合

今回テストを行い，結果的に検出された不整合は２つである．

- Transaction 記録数の制限
- int 型を超える引き出しに対する判定失敗

どちらも MakeWithdrawal に対するテストにより検出された．それぞれの不整合の詳細を述べる．

4.2.2 Transaction 記録数の制限

アカウントの引出し記録を示すレコード Account の transactions は VDM-SL 上では列として定義されており，記録できる個数は無限または未定義ということになる．一方，今回の Java での実装において，配列を用いて要素数を 100 として定義した．この 100 という数は実装にあたって恣意的に決めたものである．これにより，現状の実装では，一日の引出し額が限度を超えていない場合においても，取引がすでに 100 回記録されている場合は引出し不可能になる．

改善案

(1) 仕様を変更し，引出し回数に制限を設ける
(2) arraylist など動的に領域を確保することで実装での制限をなくす
(3) 100 回引出し記録を行なった場合に，外部にデータを書き出すなどして記録を残しつつ，新しいものに ATM 内部のデータを更新する

1 の方法は不整合はなくなるが ATM の機能として望ましくない．2 の方法だと見かけ上は無限に記録することができるが，メモリ領域の影響を受けるため，時間経過とともに，メモリを拡張していく必要がある．3 の方法も時間経過とともに外部に書き込むデータ量が増えていき，拡張の必要がある．さらに外部と通信できなければ，取引限度を超えた引出しを許す原因となる．現段階で２の方法を用いることが最適解だと考えられる．

4.2.3 int 型を超える引き出しに対する判定失敗

１日の引出限度額を 2000 に設定しているが，この引出限度額を超える引出しが可能な場合が検出された．int 型を超えるような計算が行われる場合である．この時の動作を説明する．

口座の情報（不整合に関わる部分のみを抜粋）

- 口座の残高 2147483647(int 型の限界)
- 12/23 にすでに合計額 1000 の引き出しが行われている．

この口座に対して 12/23 に 2147483645 の引き出しを行うと不整合が生じる．

期待される動作

(1) 12/23 に 2147483645 の引き出しを行う．
(2) 12/23 の引き出し合計が dailylimit を超えるため引き出しは不可能

実装の動作

(1) 12/23 に 2147483645 の引き出しを行う．
(2) 12/23 の引き出し合計が 2147483647 を超えて，負の値となる．
(3) 12/23 の引き出し合計が dailylimit を超えないため，引

き出しが行われる．

改善案

(1) 判定条件に引出し合計>0 を追加する

(2) MakeWithdrawal の事前条件として引出額<dailylimit を追加する．

仕様上 int 型を超える大きな引き出しは発生しないので，1，2の方法で十分に対処できると思われる．

4.3 考察

4.2.1 節で述べた不整合に対して，原因の考察を行なった．その結果を以下に記す．

4.3.1 Transaction 記録数の制限

仕様では transactions は列として定義されており，無限の transaction を記録可能である．しかし実装においてはメモリの領域以上の transaction を記録することはできない．今回の実装では配列を用いて要素数を 100 として定義したので記録できる transaction の上限は 100 個である．配列を用いず，arraylist などを用いて動的に領域を確保することで記録する上限を増やすことはできるが，ハードウェアの影響を受ける．

4.3.2 int 型を超える引き出しに対する判定失敗

仕様では MakeWithdrawal 内で 1 日の合計引出額が dailylimit を超えるかどうかの判定に

```
DateTotal(date,accounts(accountId).
transactions[transaction])<=dailyLimit
```

を使用している．実装では，

```
Date_total(date,accounts.
get(card.account_Id).transaction)+amo<daylimit
```

を使用している．

この時仕様では nat 型 (自然数) で実装では int 型 (整数) で計算が行われている．これにより合計引出額が対応する数を超え，負となる様な不整合が起きる．現段階では実装できていないが MakeWithdrawal に修正を加えることで防ぐことができると思われる．

5. おわりに

本研究では，仕様記述言語 VDM-SL で記述された ATM モデルに対してテストケースの作成を行い，さらに Java を用いて実装を行い実装に対しても等価なテストを適用した．そのテスト結果をもとに，仕様と実装の間での不整合を検出した．その結果を受けて不整合の原因を検出し，それを踏まえた実装の方法を提案した．

本研究では，ATM モデルに対して実装とテストを行なったが，機能が限定されているため，非常に小さい規模のモデルでの研究となった．今回検出された不整合の原因は数多くある中の一部であると考えられる．さらに大きな規模のモデルを扱うことが今後の課題となる．さらに今回行なったテストは各操作・関数を対象とした単体テストに限定されている．結合テストなどテストの幅を広げることも今後の課題である．

謝辞 本研究の一部は JSPS 科研費 24220001, 基盤研究 (S)「アーキテクチャ指向形式手法に基づく高品質ソフトウェア開発法の提案と実用化」の成果による．

参考文献

[1] 寺田夏樹. 段階的詳細化に基づく鉄道信号へのフォーマルメソッド適用法 (特集 信号通信技術). 鉄道総研報告, 2007, 21.11: 41-46.
[2] WIRTH, Niklaus. Program development by stepwise refinement. Communications of the ACM, 1971, 14.4: 221-227.
[3] Overview,http://overturetool.org,(2017/1/30 アクセス)
[4] 石川冬樹,VDM-SL,VDM++簡易リファレンス,2011, http://research.nii.ac.jp/~f-ishikawa/vdm/files/VDM-LangReference.pdf
[5] 荒木啓二郎, プログラム仕様記述論, オーム社,2002
[6] 張暁晶; 星野隆. 設計モデルを用いたテスト項目抽出とテストデータ生成手法. 電子情報通信学会技術研究報告. KBSE, 知能ソフトウェア工学, 2009, 109.41: 37-42.
[7] 佐原伸,～ソフトウェアトラブルを予防する～形式手法の技術講座, ソフト・リサーチ・センター,2008.
[8] 荒木啓二郎, プログラム仕様記述論, オーム社,2002
[9] 大西建児, et al. ソフトウェアテストの最新動向: 2. テストプロセスとテストプロセス改善. 情報処理, 2008, 49.2: 133-139.